CODE OPTIMIZATION METHOD AND SYSTEM

Arthur H. Khu

FIELD OF THE INVENTION

[0001] The invention relates generally to software code optimization and, more particularly, to a method of optimization that reduces the number of program instructions by replacing repeating patterns of code with a programming loop. It will be appreciated that the invention finds particular application to programming languages that are used to program or test programmable logic devices but also applies to any type of programming language.

BACKGROUND OF THE INVENTION

[0002] A programmable logic device (PLD) is a digital integrated circuit capable of being programmed to provide a variety of different logic functions. A PLD generally includes AND gates, OR gates, and Input/Output buffers, and functions differently depending on how it is programmed. The programming is achieved using on-chip fuses, EPROM (UV erasable programmable read-only memory) circuits, EEPROM (electrically erasable programmable read-only memory) circuits, and RAM (random access memory) circuits which programmably create data paths and logic functions within the device that are specific to the user's design.

[0003] State of the art PLDs make use of one or more non-volatile memory cell (e.g. EPROM, EEPROM, Flash EPROM, or Flash EEPROM) arrays so that they can retain their configuration memory during power-down. Typically, these arrays are erasable, thereby allowing the desired functionality of the PLD to be re-programmed many times. This programmability makes a PLD family a flexible and powerful tool for a large number of unique applications where a common PLD is employed and only the chip's internal programming is varied.

[0004] Typically, verification is required after programming to ensure that the memory cells have been properly programmed and that the PLD functions according to the design specification. Verification of a PLD involves a combination of in-circuit functional testing, simulation, timing analysis, and memory cell verification. Memory cell verification is especially important in that it determines whether the PLD has been programmed correctly according to the design specification, and thus must be performed before shipping to a user or using the device for an actual application.

[0005] In integrated circuit testing, a technique called "boundary scan" has been developed to define a standard test logic architecture for implementing boundary scan functions which can be included in an integrated circuit for the purpose of testing the integrated circuit. This standard architecture was approved on February 15, 1990 by the Institute of Electrical and Electronics Engineers (IEEE) and is commonly referred to as JTAG (Joint Test Action Group) or IEEE Standard 1149.1.

[0006] Originally, the JTAG standard created a means of verifying the integrity of traces between components on an assembled printed circuit board. This was accomplished by providing a standard architecture and a set of mandatory public instructions that all vendors claiming conformance to IEEE standard 1149.1 had to support.

[0007] Test programs can be written in accordance with the JTAG standard to program, pattern verify and functionally verify a programmable logic device. These programs can become large and can require large amounts of computer resources to store and execute. Prior art methods that attempted to optimize program code included using control-flow analysis on conditional statements such as IF-THEN and GOTO statements found in the source code or intermediate code to look for loops. Then the process attempted to create a FOR or WHILE loop by analyzing the beginning and end data conditions in the code to determine the loop index.

[0008] The prior art optimization methods provide no benefit to program source code that does not have conditional statements.

Thus, if the code did not have IF-THEN statements, a loop could not be deduced and no optimization was performed.

SUMMARY OF THE INVENTION

[0009] According to one embodiment of the present invention, a method for optimizing computer program code is provided. A keyword statement is identified and the program code is searched for the keyword statement. It is then determined if the keyword statement begins a repeating pattern of statements in the program code. The repeating pattern of statements is replaced with a program loop equivalent to the repeating pattern of statements.

[0010] According to another embodiment of the present invention, a software code optimizer is provided. The optimizer includes program instructions that analyze a software code and determine an occurrence of a repeating pattern of code therein. The optimizer also includes program instructions that convert the repeating pattern of code to a programming loop that performs an equivalent function as the repeating pattern of code.

[0011] According to yet another embodiment of the present invention, a process for optimizing software code is provided. The process locates multiple occurrences of a code pattern within the software code where the multiple occurrences appear sequentially to each other. A program loop is generated that executes one occurrence of the code pattern N times such that an equivalent result is produced as executing the multiple occurrences of the code pattern. The multiple occurrences of the code pattern are then replaced in the software code with the program loop.

[0012] One advantage of the present invention is that program code is reduced in size by replacing multiple lines of repeating code with fewer lines of a program loop code.

[0013] Still further advantages of the present invention will become apparent to those of ordinary skill in the art upon reading and understanding the following detailed description of the preferred embodiments.

BRIEF DESCRIPTION OF THE DRAWINGS

[0014] In the accompanying drawings which are incorporated in and constitute a part of the specification, embodiments of the invention are illustrated, which, together with a general description of the invention given above, and the detailed description given below, serve to example the principles of this invention.

[0015] Figure 1 is an overall system in accordance with the present invention;

[0016] Figure 2 is a flow chart for optimizing source code in accordance with the present invention; and

[0017] Figure 3 is a flow diagram for determining repeating patterns of code in accordance with the present invention.

DETAILED DESCRIPTION OF THE DRAWINGS

[0018] Illustrated in Figure 1 is an overall embodiment of the present invention. A compiler 10 is a computer program that translates a source code 15 into object code 20 that is executable by a computer. The source code 15 is a computer program written in a programming language by a programmer, automatically generated by a computer or software application, or some combination of both. The compiler 10 converts the source code 15 into assembly language and then translates the assembly language into machine language which is called the object code 20. It will be appreciated by one of ordinary skill in the art that other conversion formats and conversion steps are available depending on the programming language used. For example, the source code may be written directly as assembly language or other low-level language as is known in the art, thus, eliminating the step of converting the source code to assembly language. For purposes of the foregoing discussion, the terms "instruction" and "statement" will be used interchangeably to refer to a line of program code.

[0019] To optimize the translation process, the compiler 10 includes an optimizer 25 that analyzes the source code 15 to identify repetitive patterns of program instructions. The optimizer 25 is a function or routine of program instructions within the compiler 10. Alternately, it can be a stand-alone software application that is executable separate from the compiler 10. The optimizer 25 replaces repeating patterns of instructions with a programming loop that executes one instance of the repeating pattern multiple times. Using data arrays and appropriate loop indices, the programming loop is equivalent to the repeating pattern of instructions but requires a fewer number of instructions. Thus, after being optimized, the source code is reduced in size. Reducing the program size makes code more efficient to transfer, store and/or compile, and may enhance code performance in certain systems because of reduced memory requirements (e.g., fewer cache misses and/or page faults).

[0020] Illustrated in Figure 2 is a flow diagram of the methodology of the optimizer 25 where the blocks represent actions or events performed therein. With reference to Table 1 below, an exemplary set of computer instructions from a source code is shown. In the example, the instructions are written in the STAPL tester language which is JEDEC standard JESD71. STAPL code is used by engineers to program and test integrated chips such as programmable logic chips. To simplify the explanation, only three instructions are shown including "IRSCAN" (instruction register scan), "DRSCAN" (data register scan), and "IF-THEN" (conditional statement). Due to the large number of programming languages known in the computer arts, it will be appreciated that the invention can be applied to any programming language and any programming instructions.

[0021] With further reference to Table 1, a description of the DRSCAN instruction will be provided. Referring to line number 2 of Table 1, the DRSCAN instruction is an instruction to shift 32-bits consisting of ones "1" (represented as hexadecimal FFFFFFFF). The data read back is compared with the 32-bit hexadecimal value 09502093 masked by 0FFFFFFF. If the 32-bit data does not match, then set the "X" variable to "0". The symbol "$" identifies a data reference value represented in hexadecimal form. Here, the data references are data constants but it will be appreciated that in other languages, data variables may be used.

[0022] With further reference to Figure 2 shown at block 30, the optimizer 25 identifies one or more keywords that will be the target for optimization within the source code 15. The keyword can be predetermined within the optimizer or selected by a user on the fly. A keyword candidate is generally a source code instruction that frequently is repeated and may contain data constants. For exemplary purposes, the instruction statement "DRSCAN" is selected as the keyword.

[0023] At block 35, each instance of the keyword is found within the source code and its data constants, if present, are converted to data array references. By converting data constants to data arrays, repeating patterns become more apparent within the code and thus easier to identify. After the conversion at block 35, the source code will, for example, look like the code shown in Table 2 below. As seen in line 2, the DRSCAN statement now includes data array references dTDI, dTDO, and dMASK. It is noted that the data array references have been arbitrarily named for the present discussion and any name can be selected.

[0024] The data array references are assigned 32 array indices "0-31", shown as [31..0], to correspond to the 32-bit data constants of the original DRSCAN instruction. A similar conversion also occurs in the remaining DRSCAN statements with the only difference being that the array indices are successively incremented to store each new data constant at the next available array index values. In other words, the next available index value for the array dTDI is "32." So, the DRSCAN instruction at line 6 (a 6-bit instruction) shows array dTDI with index values 32-37. The conversion continues until all keyword instructions in the source code are converted to include data array references. After this conversion, the similar keyword instructions will look substantially identical except for their array indices.

[0025] The data array reference dTDI is defined to include the replaced data constants from the DRSCAN instructions in sequential order corresponding to the array indices assigned during conversion. An exemplary definition is shown at the bottom of Table 2 as "BOOLEAN dTDI" and the data that follows are the data constants. It will be appreciated to one of ordinary skill in the art that the syntax for defining an array is dependent on the programming language used and, thus, will change accordingly.

[0026] With reference again to Figure 2, after data references in keyword statements are converted to data array references, the optimizer 25 begins analysis of the source code to identify repeating patterns of statements including the keyword DRSCAN. At block 40, the source code is searched line by line for instances of the keyword statement. At each instance of the keyword, the optimizer determines whether a repeating pattern is present or not at block 45. At block 50, if a repeating pattern of statements is found, the repeating statements are replaced with a programming loop generated by the optimizer.

[0027] The programming loop is generated with instructions that perform an equivalent function as the replaced repeating statements but does so with fewer actual statements of code within the source code 15. An exemplary resulting programming loop is shown in Table 3 where repeating patterns of the DRSCAN statement have been replaced by a FOR-NEXT loop. The process of block 45 for determining a repeating pattern within the source code will now be explained with reference to Figure 3.

[0028] Illustrated in Figure 3 is an exemplary flow diagram of the methodology for determining repeating patterns of statements within source code in accordance with the present invention. The blocks represent actions and/or events performed therein. The methodology will be explained with reference to the exemplary program code in Table 2. To summarize, the source code is analyzed to identify and group code statements together that may form a repeating pattern of code. Locating a keyword statement signals the beginning of a first pattern. The next located keyword statement signals the end of the first pattern and the beginning of a second pattern. Of course, each keyword statement may have zero, one or more non-keyword statements that follow it which form part of their pattern. If the first and second patterns match, a repeating pattern is found that repeats at least twice. The process continues with the next keyword statement pattern and compares it to the first pattern. If it matches, the pattern now repeats three times. The process repeats. As soon as the next keyword statement pattern does not match the previous pattern, the previous pattern is converted to a programming loop (if it repeated) and the next keyword statement becomes the beginning of the next potential repeating pattern of code.

[0029] With reference to Figure 3 and Table 2, shown at block 55, the optimizer scans each line of code searching for the keyword statement, in this case, DRSCAN. The optimizer continues scanning statements, as represented by block 57, until the keyword is found. At block 60, when the keyword statement is found, it is stored in a first buffer or memory location. Although DRSCAN is found at lines 2, 6 and 8, the present description will begin with line 9 since this line starts a repeating pattern.

[0030] The statement at line 9 is stored in the first buffer. Since line 9 contains the keyword statement DRSCAN, it is the beginning of a pattern and the process determines whether the pattern repeats. At block 65, the next statement at line 10 is compared to the keyword statement in the first buffer. It is then determined, as shown at block 70, whether the next statement matches the keyword statement. Here, there is no match because line 10 does not contain the keyword. Thus, line 10 is a non-keyword statement ("NK" statement) and it is stored in the first buffer at block 75 along with the keyword statement as part of its pattern. Non-keyword statements can form part of a repeating pattern by following a keyword statement. In this manner, the pattern has the form of (1) a Keyword statement and (2) one or more non-keyword statements. The simplest form of a repeating pattern includes only repeating keyword statements with no non-keyword statements.

[0031] The process continues storing subsequent non-keyword statements in the first buffer until the next keyword statement is found. The next keyword signals the end of the pattern and the beginning of a new pattern. Shown at block 65, the next statement at line 11 is compared to the keyword statement. This time, there is a match at block 70 since line 11 contains the keyword DRSCAN. Line 11, indicated here as the second keyword statement for reference purposes, is stored in a second buffer as shown in block 80. Line 11 may be the beginning of a repeating pattern of the statements stored in the first buffer.

[0032] As represented by block 85, the process determines whether the remaining portion of the keyword statement at line 11 matches the first keyword statement at line 9. In other words, it is determined whether the data arrays of each statement match. If there is no match, then there is no occurrence of a repeating pattern at this portion of code. As shown by block 90, the first buffer is processed which includes converting any repeating patterns found into a loop form, and moving the statements contained in the second buffer into the first buffer. In other words, the keyword statement at line 11 and any subsequent non-keyword statements that have been stored in the second buffer are set as the new potential beginning of a repeating pattern. Going back to block 65, the process continues the analysis with the next statement of source code.

[0033] There are at least two ways in which to handle the optimization process. The first way is to keep the original source code as one file and then create a second file that is the optimized version of the original. In this case, when statements from the source code are analyzed and not modified, they are simply written to the optimized file in the order they appear in the original file. In the above example, when no match is found, the statements from the first buffer are written to the optimized file. When statements are optimized into looping code, the looping code is written to the optimized file replacing the original repeating statements. The final version of the optimized file is then the file that gets compiled.

[0034] The second way to handle optimization is to maintain only one file. When statements from the source code file are replaced with optimized code, they are replaced in the source code file directly. It will be appreciated that there are many ways to create and maintain data files that store and modify data temporarily or permanently as is known in the art.

[0035] With reference again to Figure 3, if the data arrays match at block 85, then a second pattern is defined in the second buffer as including the second keyword statement and adding subsequent non-keyword (NK) statements to the second buffer until another keyword statement appears in the code, or until the number of added non-keyword statements equal the number of non-keyword statements in the first buffer, which ever occurs first. For the second pattern in the second buffer to match the first pattern in the first buffer, their respective non-keyword statements must match in form and in quantity, if any exist. In the example of Table 2, line 12 is added to the second pattern and then both of the above-conditions occur. Namely, the next line 13 is a keyword statement, and only one non-keyword statement is added based on the number of non-keyword statements in the first pattern (e.g. one). Of course, there can be any number of non-keyword statements that follow a keyword including zero.

[0036] As represented by block 95, the non-keyword statements of the first and second patterns are compared. If there is a match at block 100, a repeating pattern is found as shown at block 105. The process continues at block 110 by determining if there are additional instances of the repeating pattern that follow. This process repeats until a keyword statement is found that does not match the first keyword statement of the pattern or if the non-keyword statements do not match those which follow the first keyword statement. When such a non-match is found, shown at block 90, the pattern of statements in the first buffer are converted to loop form if they were found to repeat, the first buffer is reset, and the statements in the second buffer are moved to the first buffer, and the process continues.

[0037] With further reference to Table 2, the methodology of determining a data array match shown at block 85 will be described. When comparing keyword statements at line 9 and line 11, each statement has the same data array references, namely, dTDI, dTDO and dMASK. However, the array indices do not match identically. This condition does not eliminate the possibility of a match. Rather, it is determined whether the size of the data arrays match and whether the difference in index values match. In other words, if the number of index values within each data array reference is identical in both keyword statements and if the index values are sequential to each other, the keyword statements match.

[0038] Looking to the keyword statement at line 9, the data array reference dTDI has index values 65 to 91 which is a size of 27 index values. Looking to the keyword statement at line 11, data array reference dTDI has index values 92 to 118 which is a size of 27 index values. Since both data array references have the same number of index values, the data arrays match in size. The same comparison is then performed for the other data array references for each keyword statement.

[0039] Then it is determined if the data array index values are sequential to each other. This is performed by, for example, finding the difference between the first and last index values from the two data arrays. Comparing the dTDI array of line 9 and the dTDI array of line 11, the difference between the first index values is 118-91 = 27. The difference between the last index values is 92-65 = 27. Since the differences are equal, it indicates that the index values of the two dTDI arrays are sequential and, thus, qualify for a repeating pattern. Of course, determining if indices are sequential will depend on the programming language being analyzed and its syntax for arrays.

[0040] With reference again to Table 2, by continuing the analysis of the program code, the pattern defined by the DRSCAN/IF-THEN statements of lines 9 and 10 is found to repeat six (6) times from line 9 to line 20 of the program code. This repeating pattern of code is then optimized by converting it to a loop statement/instruction including an appropriate loop instruction and one instance of the repeating pattern. The loop is set to execute the pattern the same number of times that the pattern was found in the program code. In this example, the number is six (6). The data array index values in the pattern are modified so that each iteration of the loop addresses the next sequence of data array index values. An exemplary programming loop is shown in Table 3 at lines 9-12. This loop, thus, replaces the repeating pattern of code and reduces the size of the program code.

[0041] With reference to Table 3, an exemplary FOR-NEXT loop is used as the replacement code. A loop variable iVar is defined from 0 to 5 so that the loop iterates six (6) times which is the number of instances of the repeating pattern found. The DRSCAN statement is modified to include the loop variable iVar in its array index so that each iteration of the FOR-NEXT loop references the next sequential DRSCAN data values from the original program code. In this case, the loop variable iVar is multiplied by 27 which is the number of index values in each data array reference. It is also seen that the number 91 and 65 are added to the index variables. These numbers represent the first and last index values from the first instance of the repeating pattern found at line 9 of the original program code. Similar index values are also used for the remaining data array references. The remaining non-keyword statements in the pattern, in this case the IF-THEN statement, follow the keyword statement in the FOR-NEXT loop.

[0042] It will be appreciated by one of ordinary skill in the art that the array index variables will depend on the data array structure and index value range found in the repeating pattern. Also, the programming loop syntax will depend on the source code language used and the desired technique to performing the loop. Depending on the type of loop and technique, the data array index values will change accordingly as will be easily determined by one of ordinary skill. Exemplary types of loops or looping instructions that may be used with the present process but not limited thereto include FOR loops, WHILE loops, DO loops, and loops controlled by branching statements like GOTO, JUMP, BRANCH, etc.

[0043] With the present invention, source code can be reduced in size by replacing repeating patterns of statements with equivalent program loop code. Compiling time may be reduced and more efficient object code can be generated.

[0044] To further optimize the source code, the process of the present invention can be applied multiple times. In this manner, a different keyword statement is selected and the analysis is performed again. The process can also be performed to optimize the "optimized" code. To perform this optimization, the analysis targets the looping instruction to determine repeating patterns that may have been generated from a previous optimization.

[0045] While the present invention has been illustrated by the description of embodiments thereof, and while the embodiments have been described in considerable detail, it is not the intention of the applicants to restrict or in any way limit the scope of the appended claims to such detail. Additional advantages and modifications will readily appear to those skilled in the art. For example, the present invention can apply to any type of source code or programming language instead of the exemplary program language shown above. Additionally, the optimization process can use more than one keyword when searching for repeating patterns of code. Therefore, the invention, in its broader aspects, is not limited to the specific details, the representative apparatus, and illustrative examples shown and described. Accordingly, departures may be made from such details without departing from the spirit or scope of the applicant's general inventive concept.

<u>Table 1 - Sample Source Code</u>

```
IRSCAN 8, $FE; ←Instruction Register SCAN 8 bits of data (hex FE)
DRSCAN 32, $FFFFFFFF,COMPARE $09502093,$0FFFFFFF,X;
IF (!X) THEN GOTO FAILURE;
IRSCAN 8, $FF;
IRSCAN 8, $E8;
DRSCAN 6, $07;  ← Data Register SCAN 6 bits of data (hex 07)
IRSCAN 8, $EE;
DRSCAN 27, $00003FE;
DRSCAN 27, $00007FE,COMPARE $00003FF,$7FFFFFF,X; ←repeating pattern
IF (!X) THEN GOTO FAILURE;                       ←line 2 of pattern
DRSCAN 27, $0000BFE,COMPARE $00007FF,$7FFFFFF,X;
IF (!X) THEN GOTO FAILURE;
DRSCAN 27, $0000FFE,COMPARE $0000BFF,$7FFFFFF,X;
IF (!X) THEN GOTO FAILURE;
DRSCAN 27, $00013FE,COMPARE $0000FFF,$7FFFFFF,X;
IF (!X) THEN GOTO FAILURE;
DRSCAN 27, $00023FE,COMPARE $00013FF,$7FFFFFF,X;
IF (!X) THEN GOTO FAILURE;
DRSCAN 27, $00027FE,COMPARE $00023FF,$7FFFFFF,X;
IF (!X) THEN GOTO FAILURE;
```

Table 2 - Source Code After Data Array Conversion

```
IRSCAN 8, $FE;
DRSCAN 32,dTDI[31..0],COMPARE dTDO[31..0],dMASK[31..0],X;
IF (!X) THEN GOTO FAILURE;
IRSCAN 8, $FF;
IRSCAN 8, $E8;
DRSCAN 6,dTDI[37..32];
IRSCAN 8, $EE;
DRSCAN 27,dTDI[64..38];
DRSCAN27,dTDI[91..65],COMPARE dTDO[58..32],dMASK[58..32],X;
IF (!X) THEN GOTO FAILURE;
DRSCAN27,dTDI[118..92],COMPARE dTDO[85..59],dMASK[85..59],X;
IF (!X) THEN GOTO FAILURE;
DRSCAN 27,dTDI[145..119],COMPARE dTDO[112..86],dMASK[112..86],X;
IF (!X) THEN GOTO FAILURE;
DRSCAN 27,dTDI[172..146],COMPARE dTDO[139..113],dMASK[139..113],X;
IF (!X) THEN GOTO FAILURE;
DRSCAN 27,dTDI[199..173],COMPARE dTDO[166..140],dMASK[166..140],X;
IF (!X) THEN GOTO FAILURE;
DRSCAN 27,dTDI[226..200],COMPARE dTDO[193..167],dMASK[193..167],X;
IF (!X) THEN GOTO FAILURE;

BOOLEAN dTDI[68108] =  $03…    ← data array definition
…
91FF00113FE0021FFC0042FF80083FF00103FE
001A7FC0033FF80065FF000C7FE00187FC002CFF80057FF000ABFE0014FFC0028
FF80049FF0008FFE00117FC0021FF80041FF00053FE0009FFC0012FF80023FF00
043FE00067FC000BFF80015FF00027FE00047FC0004FF80007FF0000BFE0000FF
C0000FF87FFFFFFFF;
```

Table 3 - Optimized Source Code

```
IRSCAN 8, $FE;
DRSCAN 32,dTDI[31..0],COMPARE dTDO[31..0],dMASK[31..0],X;
IF (!X) THEN GOTO FAILURE;
IRSCAN 8, $FF;
IRSCAN 8, $E8;
DRSCAN 6,dTDI[37..32];
IRSCAN 8, $EE;
DRSCAN 27,dTDI[64..38];
FOR iVar = 0 to 5;
DRSCAN 27, dTDI[(iVar*27)+91..(iVar*27)+65],
    COMPARE dTDO[(iVar*27)+58..(iVar*27)+32],
    dMASK[(iVar*27)+58..(iVar*27)+32],X;
IF (!X) THEN GOTO FAILURE;
NEXT iVar;
```